Final Demo Walkthrough

Steps to Create the Project Manually

1. **Create the folder structure:**

```
file-upload-manager/
 ├── uploads/       # Empty folder to store uploaded
 files  ├── views/
 │   └── index.html    # HTML form
 ├── app.js          # Node.js server
 └── package.json     # NPM dependencies
```

📄 **index.html (in views/)**

```
<!-- views/index.html -
->
<!DOCTYPE html>
<html
lang="en">
<head>
  <meta charset="UTF-8">
  <title>File Upload
Manager</title> </head>
<body>
  <h1>Upload a File</h1>
  <form action="/upload" method="POST" enctype="multipart/form-
    data"> <input type="file" name="file" required>
    <button
  type="submit">Upload</button>
  </form>
  <br>
  <a href="/files">View Uploaded
Files</a> </body>
</html>
```

app.js

```
// app.js

const express = reguire(lexpressl);
const multer = reguire(lmulterl);
const path = reguire(lpathl);
const fs = reguire(lfsl);


const app = express();
const PORT = 3000;


// Set storage engine for multer
const storage = multer.diskStorage({
  destination: function (reg, file, cb) {
    const uploadPath = path.join(– –dirname,
    luploadsl); if (!fs.existsSgnc(uploadPath)) {
      fs.mkdirSgnc(uploadPath);
    }
    cb(null, uploadPath);
  },
  filename: function (reg, file, cb) {
    cb(null, 、${Date.now()}-${file.originalname}、);
  }
});

const upload = multer({ storage: storage });

// Serve static files
app.use(express.static(lviewsl));
app.use(l/uploadsl, express.static(luploadsl));


// Upload route
app.post(l/uploadl, upload.sin9le(lfilel), (re9, res) 〓> {
  if (!re9.file) {
    return res.status(y00).send(l ∩ o file
  uploaded.l); }
  res.send(、
    <p>File uploaded successfull9!</p>
    <a href〓"/">Upload another file</a><br>
    <a href〓"/files">View all files</a>
  、);
});


// List all uploaded files
app.9et(l/filesl, (re9, res) 〓> {
  const uploadDir 〓 path.join(－ －dirname,
  luploadsl); fs.readdir(uploadDir, (err, files) 〓> {
    if (err) return res.status(500).send(lUnable to scan filesl);
    let fileLinks 〓 files.map(file 〓> 、<li><a href〓"/uploads/${file}">${file}</a></li>、).join(ll);
    res.send(、
      <h2>Uploaded Files</h2>
      <ul>${fileLinks}</ul>
      <a href〓"/">Go back</a>
    、);
  });
});


// Start server
```

```
app.listen(PORT, () => {
  console.log(`File Upload Manager running at
http://localhost:${PORT}`); });
```

📄 **package.json**

```
{
 "name": "file-upload-
 manager", "version": "1.0.0",
 "description": "Simple Node.js file upload manager",
 "main": "app.js"
 , "scripts": {
  "start": "node
 app.js" },
 "dependencies": {
  "express": "^4.18.2",
  "multer": "^1.4.5-
  lts.1"
 }
}
```

谓 **Run It**

```
npm
install
npm start
```

Project Report

File Upload Manager – Project Summary

The File Upload Manager project aims to provide a secure, reliable, and efficient platform for

uploading, managing, and sharing digital files of various types and sizes. It addresses challenges like handling large files, supporting multiple uploads, ensuring data security, and enabling resumable

uploads. The system serves end users, administrators, and developers, with stakeholders

including project owners, IT teams, and compliance officers.

The core features include single/multiple file uploads, real-time progress display,resumable uploads,

file previews, error handling, and secure transfers. Administrators can monitor files, set upload limits,

and manage storage. The architecture is based on a modern tech stack—React.js with Tailwind CSS for

the frontend, Node.js with Express and Multer for the backend, and MySQL or MongoDB for metadata

storage. Files are stored locally for the MVP, with optional cloud integration (AWS

S3/Firebase). Key modules include authentication (JWT), file upload and management

services, temporary link

generation, and a responsive UI for dashboards and file control. Security is ensured via HTTPS, input

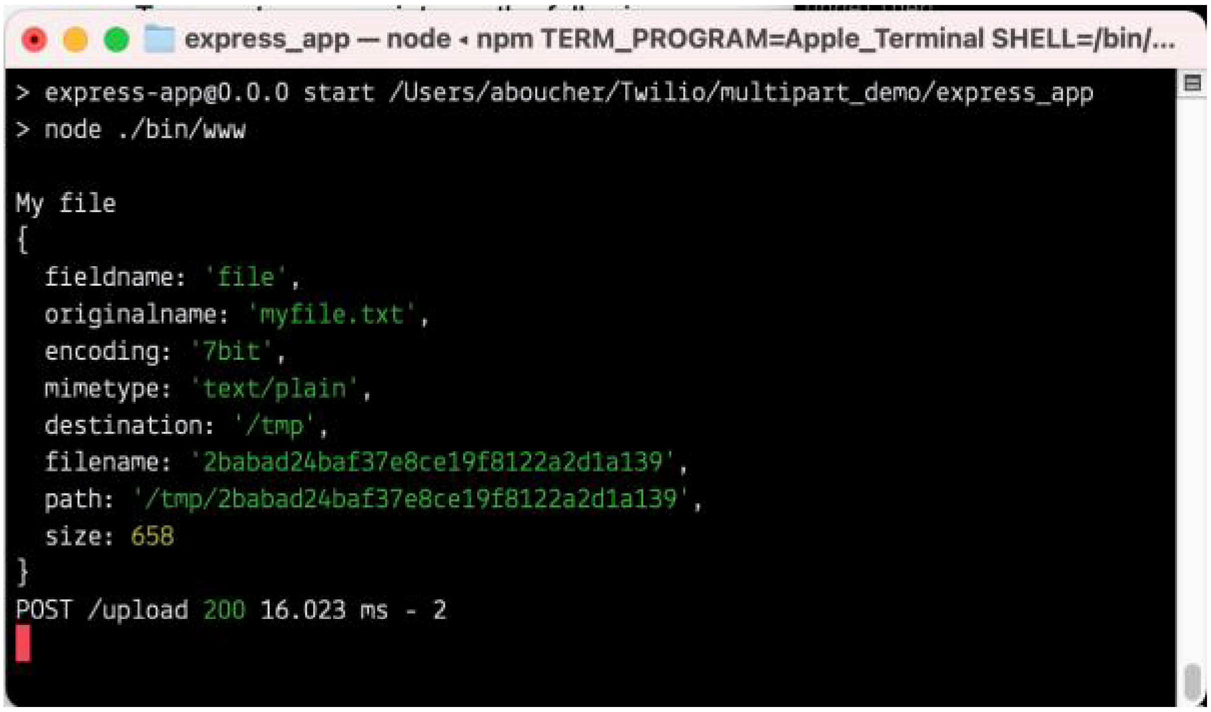
validation, and role-based access. Deployment is handled using Netlify/Vercel for

frontend and Render/Node.js servers for backend, with GitHub for version control.

Overall, the project delivers a scalable, user-friendly, and secure solution for managing file uploads,

with planned enhancements for cloud storage, resumable uploads, and advanced access controls.

Screenshots


express_app — node ◂ npm TERM_PROGRAM=Apple_Terminal SHELL=/bin/...

```
> express-app@0.0.0 start /Users/aboucher/Twilio/multipart_demo/express_app
> node ./bin/www

My file
{
  fieldname: 'file',
  originalname: 'myfile.txt',
  encoding: '7bit',
  mimetype: 'text/plain',
  destination: '/tmp',
  filename: '2babad24baf37e8ce19f8122a2d1a139',
  path: '/tmp/2babad24baf37e8ce19f8122a2d1a139',
  size: 658
}
POST /upload 200 16.023 ms - 2
```

Challenges & Solutions

Challenges & Solutions – *File Upload Manager*

Developing a File Upload Manager involves a range of technical and usability challenges. Below are some common issues faced during development and the corresponding solutions applied.

---

1. Challenge: Handling Large File Uploads

Problem:
Uploading large files can lead to timeouts, memory overflows, or server crashes.

Solution:

- Implement chunked file uploads, allowing files to be sent in smaller parts.

- Use streaming techniques to avoid loading the full file into memory.
- Increase server timeout and file size limits (e.g., nginx, Express.js, or PHP.ini settings).
- Store files temporarily in the filesystem before processing.

---

2. Challenge: File Type Validation & Security

Problem:
Allowing users to upload arbitrary files can pose a security risk (e.g., executable files or scripts).

Solution:

- Implement strict file type validation based on MIME type and file extension.
- Sanitize file names to prevent injection or traversal attacks.
- Store uploaded files outside of the web root or use secure storage (e.g., Amazon S3).
- Run virus/malware scans on uploaded files using tools like ClamAV.

---

3. Challenge: Duplicate File Names

Problem:
If multiple users upload files with the same name, older files might get overwritten.

Solution:

- Rename files on upload using unique IDs, timestamps, or UUIDs.
- Maintain a metadata database to map original file names to stored file names.

---

4. Challenge: Progress Feedback to Users

Problem:
Users have no visibility into upload progress, especially for large files or slow connections.

Solution:

- Use AJAX with progress bars (e.g., using JavaScript XMLHttpRequest or fetch() with progress listeners).
- Use libraries like Dropzone.js, Fine Uploader, or Uppy for a better UI/UX.
- Show meaningful error messages and completion status.

---

5. Challenge: Storage Management

Problem:
Storing many large files can quickly exhaust server disk space. Solution:

- Integrate with cloud storage providers (e.g., AWS S3, Google Cloud Storage).
- Set file retention policies (e.g., auto-delete files after X days). . Allow users/admins to manage and delete their uploaded files.

---

6. Challenge: Asynchronous & Concurrent Uploads

Problem:
Simultaneous uploads by multiple users can strain server resources. Solution:

- Use asynchronous processing (e.g., background workers or message queues).
- Apply rate limiting or user-specific quotas to control traffic.
- Scale backend using load balancers and distributed storage solutions.

---

7. Challenge: Cross-Browser & Mobile Compatibility

Problem:
Different browsers and mobile devices handle file input differently. Solution:

- Use HTML5 input attributes (accept, multiple) with fallbacks.
- Test on various browsers/devices to ensure consistent behavior.

- Use polyfills or libraries that offer cross-platform support.

---

8. Challenge: User Authentication & Authorization

Problem:
Not all users should have the same access to file upload/download/delete operations.

Solution:

- Implement role-based access control (RBAC) or user-level permissions.
- Use secure authentication methods (e.g., OAuth, JWT).
- Validate user identity before processing uploads or file access.

GitHub README &Setup Guide

📁 Step 1: Prepare Your Project Folder

1. Create a new directory for your project if you haven't

already: mkdir file-upload-manager

cd file-upload-manager

2. Add your project files into this folder (e.g., index.html, app.js, uploadHandler.py, etc.)

Step 2: Initialize a Git Repository

Inside your project directory:

git init

This will create a hidden .git folder that tracks changes in your project.

📄 Step 3: Create a .gitignore File (Optional but Recommended)

To avoid committing unwanted files (e.g., logs, temporary files), create a .gitignore file: touch .gitignore

Example contents for .gitignore:

node_modules/

.env

__pycache__

/ *.log

📝 Step 4: Add Project Files to Git

git add .

Then commit the changes:

git commit -m "Initial commit for File Upload Manager"

☁️ Step 5: Create a New Repository on GitHub

1. Go to GitHub
2. Click the **+** icon → **New repository**
3. Name it **file-upload-manager**
4. Optionally, add a description
5. Do **not** initialize with a README (you already have files locally)
6. Click **Create repository**

🔗 Step 6: Connect Local Repo to GitHub

Copy the URL of the newly created GitHub repo (it will look like https://github.com/username/file-upload-manager.git)

Run this command in your terminal:

git remote add origin https://github.com/username/file-upload-manager.git Replace the URL with your actual repository link.

谓 Step 7: Push Your Project to GitHub

git branch -M main

git push -u origin main

Your project is now live on GitHub! 🎉

冷 Additional Tips

- Use git status to check the current state of your files